Lecture: Manycore GPU Architectures and Programming, Part 2

CSCE 569 Parallel Computing

Department of Computer Science and Engineering
Yonghong Yan
yanyh@cse.sc.edu
https://passlab.github.io/CSCE569/
Manycore GPU Architectures and Programming: Outline

• Introduction
  – GPU architectures, GPGPUs, and CUDA

• GPU Execution model

CUDA Programming model
• Working with Memory in CUDA
  – Global memory, shared and constant memory

• Streams and concurrency

• CUDA instruction intrinsic and library

• Performance, profiling, debugging, and error handling

• Directive-based high-level programming model
  – OpenACC and OpenMP
How is the GPU controlled?

• The CUDA API is split into:
  – The CUDA Management API
  – The CUDA Kernel API

• The CUDA Management API is for a variety of operations
  – GPU memory allocation, data transfer, execution, resource creation
  – Mostly regular C function and calls

• The CUDA Kernel API is used to define the computation to be performed by the GPU
  – C extensions
How is the GPU controlled?

• A CUDA kernel:
  – Defines the operations to be **performed by a single thread** on the GPU
  – Just as a C/C++ function defines work to be done on the CPU
  – Syntactically, a kernel looks like C/C++ with some extensions

  ```
  __global__ void kernel(...) {
    ...
  }
  ```

  – Every CUDA thread executes the same kernel logic (SIMT)
  – Initially, the only difference between threads are their **thread coordinates**
How are GPU threads organized?

- **CUDA thread hierarchy**
  - **Warp** = SIMT Group
  - **Thread Block** = SIMT Groups that run concurrently on an SM
  - **Grid** = All Thread Blocks created by the same kernel launch

- Launching a kernel is simple and similar to a function call.
  - kernel name and arguments
  - # of thread blocks/grid and # of threads/block to create:

```c
kernel<<<nbblocks,
    threads_per_block>>>(arg1, arg2, ...);
```
How are GPU threads organized?

• In CUDA, only thread blocks and grids are first-class citizens of the programming model.

• The number of warps created and their organization are implicitly controlled by the kernel launch configuration, but never set explicitly.

```
kernel<<<nbblocks,
threads_per_block>>>(arg1, arg2, ...);
```

kernel launch configuration
How are GPU threads organized?

- GPU threads can be configured in one-, two-, or three-dimensional layouts

  - One-dimensional blocks and grids:
    ```
    int nblocks = 4;
    int threads_per_block = 8;
    kernel<<<nbblocks, threads_per_block>>>(...);
    ```
How are GPU threads organized?

- GPU threads can be configured in one-, two-, or three-dimensional layouts

  - Two-dimensional blocks and grids:
    ```
    dim3 nblocks(2,2)
    dim3 threads_per_block(4,2);
    kernel<<<nblocks, threads_per_block>>>(...);
    ```
How are GPU threads organized?

• GPU threads can be configured in one-, two-, or three-dimensional layouts

  – Two-dimensional grid and one-dimensional blocks:
    
    ```
    dim3 nblocks(2,2);
    int threads_per_block = 8;
    kernel<<<nbblocks, threads_per_block>>>(...);
    ```
How are GPU threads organized?

- On the GPU, the number of blocks and threads per block is exposed through intrinsic thread coordinate variables:
  
  - **Dimensions**
  - **IDs**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gridDim.x, gridDim.y, gridDim.z</code></td>
<td>Number of blocks in a kernel launch.</td>
</tr>
<tr>
<td><code>blockIdx.x, blockIdx.y, blockIdx.z</code></td>
<td>Unique ID of the block that contains the current thread.</td>
</tr>
<tr>
<td><code>blockDim.x, blockDim.y, blockDim.z</code></td>
<td>Number of threads in each block.</td>
</tr>
<tr>
<td><code>threadIdx.x, threadIdx.y, threadIdx.z</code></td>
<td>Unique ID of the current thread within its block.</td>
</tr>
</tbody>
</table>
How are GPU threads organized?

to calculate a **globally unique ID** for a thread on the GPU inside a one-dimensional grid and one-dimensional block:

```c
__global__ void kernel(...) {
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  ...
}
```

Block 0  Block 1  Block 2  Block 3

- blockIdx.x = 2;
- blockDim.x = 8;
- threadIdx.x = 2;
How are GPU threads organized?

• Thread coordinates offer a way to differentiate threads and identify thread-specific input data or code paths.
  – Link data and computation, a mapping

```c
__global__ void kernel(int *arr) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < 32) {
        arr[tid] = f(arr[tid]);
    } else {
        arr[tid] = g(arr[tid]);
    }
}
```

Thread Divergence: recall that useless code path is executed, but then disabled in SIMT execution model
How is GPU memory managed?

- CUDA Memory Management API
  - Allocation of GPU memory
  - Transfer of data from the host to GPU memory
  - Free-ing GPU memory
  - Foo(int A[][N]) {
  }

<table>
<thead>
<tr>
<th>Host Function</th>
<th>CUDA Analogue</th>
</tr>
</thead>
<tbody>
<tr>
<td>malloc</td>
<td>cudaMalloc</td>
</tr>
<tr>
<td>memcpy</td>
<td>cudaMemcpy</td>
</tr>
<tr>
<td>free</td>
<td>cudaFree</td>
</tr>
</tbody>
</table>
How is GPU memory managed?

cudaError_t cudaMalloc(void **devPtr, size_t size);
  – Allocate size bytes of GPU memory and store their address at *devPtr

cudaError_t cudaFree(void *devPtr);
  – Release the device memory allocation stored at devPtr
  – Must be an allocation that was created using cudaMalloc
How is GPU memory managed?

cudaError_t cudaMemcpy(
    void **dst, const void **src, size_t count,
    enum cudaMemcpyKind kind);
    – Transfers count bytes from the memory pointed to by src to dst
    – kind can be:
      • cudaMemcpyHostToHost,
      • cudaMemcpyHostToDevice,
      • cudaMemcpyDeviceToHost,
      • cudaMemcpyDeviceToDevice
    – The locations of dst and src must match kind, e.g. if kind is cudaMemcpyHostToDevice then src must be a host array and dst must be a device array
How is GPU memory managed?

```c
void *d_arr, *h_arr;
h_addr = ...; /* init host memory and data */
// Allocate memory on GPU and its address is in d_arr
cudaMalloc((void **) &d_arr, nbytes);

// Transfer data from host to device
cudaMemcpy(d_arr, h_arr, nbytes,
            cudaMemcpyHostToDevice);

// Transfer data from a device to a host
cudaMemcpy(h_arr, d_arr, nbytes,
            cudaMemcpyDeviceToHost);

// Free the allocated memory
cudaFree(d_arr);
```
CUDA Program Flow

• At its most basic, the flow of a CUDA program is as follows:
  1. Allocate GPU memory
  2. Populate GPU memory with inputs from the host
  3. Execute a GPU kernel on those inputs
  4. Transfer outputs from the GPU back to the host
  5. Free GPU memory

• Let’s take a look at a simple example that manipulates data
AXPY Example with OpenMP: Multicore

- $y = \alpha \cdot x + y$
  - $x$ and $y$ are vectors of size $n$
  - $\alpha$ is scalar

```c
void axpy(REAL *x, REAL *y, long n, REAL a) {
  #pragma omp parallel for shared(x, y, n, a)
  for (int i = 0; i < n; ++i)
    y[i] += a * x[i];
}
```

- Data ($x$, $y$ and $a$) are shared
  - Parallelization is relatively easy
CUDA Program Flow

• AXPY is an **embarrassingly parallel problem**
  – How can vector addition be parallelized?
  – How can we map this to GPUs?

• Each thread does one element
AXPY Offloading To a GPU using CUDA

```c
#include <stdio.h>
#include <cuda_runtime.h>

__global__ void axpy(REAL *x, REAL *y, int n, REAL a) {
    int id = blockIdx.x*blockDim.x+ threadIdx.x;
    if (id < n) y[id] += a * x[id];
}

int main(int argc, char* argv[]) {
    // ... init host a, x and y
    // Allocate memory for each vector on GPU
    cudaMalloc(&d_x, size);
    cudaMalloc(&d_y, size);

    // Copy host vectors to device
    cudaMemcpy( d_x, h_x, size, cudaMemcpyHostToDevice);
    cudaMemcpy( d_y, h_y, size, cudaMemcpyHostToDevice);

    int blockSize, gridSize;
    blockSize = 1024;
    gridSize = (int)ceil((float)n/blockSize);
    axpy<<<gridSize, blockSize>>>(d_x, d_y, n, a);

    // Copy array back to host
    cudaMemcpy( h_y, d_y, size, cudaMemcpyDeviceToHost );

    // Release device memory
    cudaFree(d_x);
    cudaFree(d_y);
}
```
CUDA Program Flow

• Consider the workflow of the example vector addition \texttt{vecAdd.cu}:
  1. Allocate space for A, B, and C on the GPU
  2. Transfer the initial contents of A and B to the GPU
  3. Execute a kernel in which each thread sums $A_i$ and $B_i$, and stores the result in $C_i$
  4. Transfer the final contents of C back to the host
  5. Free A, B, and C on the GPU

Modify to $C = A+B+C$

\begin{align*}
A &= B*C; \\
\text{we will need both C and A in the host side after GPU computation.}
\end{align*}

• Compile and running on bridges:
  – https://passlab.github.io/CSCE569/resources/HardwareSoftware.html#interactive
  – copy \texttt{gpu_code_examples} folder from my home folder
  • \texttt{cp \textendash r \tilde{yan}/gpu_code_examples \tilde{yan}}
  – $\texttt{nvcc \textendash Xcompiler \textendash fopenmp vectorAdd.cu}$
  – $\texttt{./a.out}$
More Examples and Exercises

• Matvec:
  – Version 1: each thread computes one element of the final vector
  – Version 2:

• Matmul in assignment #4
  – Version 1: each thread computes one row of the final matrix C
CUDA SDK Examples

• CUDA Programming Manual:

• CUDA SDK Examples on bridges
  – module load gcc/5.3.0 cuda/8.0
  – export CUDA_PATH=/opt/packages/cuda/8.0
  – /opt/packages/cuda/8.0/samples

• Copy to your home folder
  – cp –r /opt/packages/cuda/8.0/samples ~/CUDA_samples

• Do a “make” in the folder, and it will build all the sources

• Or go to a specific example folder and make, it will build only the binary

• Find ones you are interested in and run to see
Inspecting CUDA Programs

• Debugging CUDA program:
  – `cuda-gdb` debugging tool, like `gdb`

• Profiling a program to examine the performance
  – `Nvprof tool, like gprof`
  – `Nvprof ./vecAdd`
Manycore GPU Architectures and Programming: Outline

• Introduction
  – GPU architectures, GPGPUs, and CUDA
• GPU Execution model
• CUDA Programming model

Working with Memory in CUDA
  – Global memory, shared and constant memory

• Streams and concurrency
• CUDA instruction intrinsic and library
• Performance, profiling, debugging, and error handling
• Directive-based high-level programming model
  – OpenACC and OpenMP
Storing Data on the GPU

• A memory hierarchy emulates a large amount of low-latency memory
  – Cache data from a large, high-latency memory bank in a small low-latency memory bank
Storing Data on the GPU

- The CUDA Memory Hierarchy is more complex than the CPU memory hierarchy
  - Many different types of memory, each with special-purpose characteristics
  - More explicit control over data movement
Storing Data on the GPU

- Registers
  - Lowest latency memory space on the GPU
  - **Private to each CUDA thread**
  - Constant pool of registers per-SM divided among threads in resident thread blocks
  - Architecture-dependent limit on number of registers per thread
  - Registers are not explicitly used by the programmer, implicitly allocated by the compiler
  - `-maxrregcount` compiler option allows you to limit # registers per thread
Storing Data on the GPU

• GPU Caches
  – Behaviour of GPU caches is architecture-dependent
  – Per-SM L1 cache stored on-chip
  – Per-GPU L2 cache stored off-chip, caches values for all SMs
  – Due to parallelism of accesses, GPU caches do not follow the same LRU rules as CPU caches
Storing Data on the GPU

• Shared Memory
  – **Declared with the __shared__ keyword**
  – Low-latency, high bandwidth
  – **Shared by all threads in a thread block**
  – Explicitly allocated and managed by the programmer, manual L1 cache
  – Stored on-SM, same physical memory as the GPU L1 cache
  – On-SM memory is statically partitioned between L1 cache and shared memory
Storing Data on the GPU

- **Constant Memory**
  - Declared with the `__constant__` keyword
  - Read-only
  - Limited in size: 64KB
  - Stored in device memory (same physical location as Global Memory)
  - Cached in a per-SM constant cache
  - Optimized for all threads in a warp accessing the same memory cell
Storing Data on the GPU

• Texture Memory
  – **Read-only**
  – **Stored in device memory (same physical location as Global Memory)**
  – Cached in a texture-only on-SM cache
  – Optimized for 2D spatial locality (caches commonly only optimized for 1D locality)
  – Explicitly used by the programmer
  – Special-purpose memory
Storing Data on the GPU

- **Global Memory**
  - Large, high-latency memory
  - Stored in device memory (along with constant and texture memory)
  - Can be declared statically with `__device__`
  - Can be allocated dynamically with `cudaMalloc`
  - Explicitly managed by the programmer
  - Optimized for all threads in a warp accessing neighbouring memory cells
# Storing Data on the GPU

<table>
<thead>
<tr>
<th>MEMORY</th>
<th>ON/OFF CHIP</th>
<th>Cached</th>
<th>ACCESS</th>
<th>SCOPE</th>
<th>LIFETIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>On</td>
<td>n/a</td>
<td>R/W</td>
<td>1 thread</td>
<td>Thread</td>
</tr>
<tr>
<td>Local</td>
<td>Off</td>
<td>†</td>
<td>R/W</td>
<td>1 thread</td>
<td>Thread</td>
</tr>
<tr>
<td>Shared</td>
<td>On</td>
<td>n/a</td>
<td>R/W</td>
<td>All threads in block</td>
<td>Block</td>
</tr>
<tr>
<td>Global</td>
<td>Off</td>
<td>†</td>
<td>R/W</td>
<td>All threads + host</td>
<td>Host allocation</td>
</tr>
<tr>
<td>Constant</td>
<td>Off</td>
<td>Yes</td>
<td>R</td>
<td>All threads + host</td>
<td>Host allocation</td>
</tr>
<tr>
<td>Texture</td>
<td>Off</td>
<td>Yes</td>
<td>R</td>
<td>All threads + host</td>
<td>Host allocation</td>
</tr>
</tbody>
</table>
# Storing Data on the GPU

<table>
<thead>
<tr>
<th>QUALIFIER</th>
<th>VARIABLE NAME</th>
<th>MEMORY</th>
<th>SCOPE</th>
<th>LIFESPAN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>float var</td>
<td>Register</td>
<td>Thread</td>
<td>Thread</td>
</tr>
<tr>
<td>float</td>
<td>var[100]</td>
<td>Local</td>
<td>Thread</td>
<td>Thread</td>
</tr>
<tr>
<td><strong>shared</strong></td>
<td>float var†</td>
<td>Shared</td>
<td>Block</td>
<td>Block</td>
</tr>
<tr>
<td><strong>device</strong></td>
<td>float var†</td>
<td>Global</td>
<td>Global</td>
<td>Application</td>
</tr>
<tr>
<td><strong>constant</strong></td>
<td>float var†</td>
<td>Constant</td>
<td>Global</td>
<td>Application</td>
</tr>
</tbody>
</table>
Concentrate On:

- Global memory
- Shared memory
- Constant memory
Static Global Memory

• Static Global Memory has a fixed size throughout execution time:

  ```c
  __device__ float devData;
  __global__ void checkGlobalVariable()
  {
    printf("devData has value %f\n", devData);
  }
  ```

• Initialized using `cudaMemcpyToSymbol`:

  ```c
  cudaMemcpyToSymbol(devData, &hostData, sizeof(float));
  ```

• Fetched using `cudaMemcpyFromSymbol`:

  ```c
  cudaMemcpyFromSymbol(&hostData, devData, sizeof(float));
  ```
Dynamic Global Memory

• We have already seen dynamic global memory
  – `cudaMalloc` dynamically allocates global memory
  – `cudaMemcpy` transfers to/from global memory
  – `cudaFree` frees global memory allocated by `cudaMalloc`

• `cudaMemcpy` supports 4 types of transfer:
  – `cudaMemcpyHostToHost`,
    `cudaMemcpyHostToDevice`,
    `cudaMemcpyDeviceToHost`,
    `cudaMemcpyDeviceToDevice`

• You can also memset global memory

```c
cudaError_t cudaMemcpy(void *devPtr, int value, size_t count);
```
Global Memory Access Patterns

• CPU caches are optimized for linear, iterative memory accesses
  – Cache lines ensure that accessing one memory cell brings neighbouring memory cells into cache
  – If an application exhibits good spatial or temporal locality (which many do), later references will also hit in cache
Global Memory Access Patterns

• GPU caching is a more challenging problem
  – Thousands of threads cooperating on a problem
  – Difficult to predict the next round of accesses for all threads

• For efficient global memory access, GPUs instead rely on:
  – Large device memory bandwidth
  – Aligned and coalesced memory access patterns
  – Maintaining sufficient pending I/O operations to keep the memory bus saturated and hide global memory latency
Global Memory Access Patterns

- Achieving **aligned** and **coalesced** global memory accesses is key to optimizing an application’s use of global memory bandwidth.

  - **Coalesced**: the threads within a warp reference memory addresses that can all be serviced by a single global memory transaction (think of a memory transaction as the process of bring a cache line into the cache).

  - **Aligned**: the global memory accesses by threads within a warp start at an address boundary that is an even multiple of the size of a global memory transaction.
Global Memory Access Patterns

• A global memory transaction is either 32 or 128 bytes
  – The size of a memory transaction depends on which caches it passes through
  – If L1 + L2: 128 byte
  – If L2 only: 32 bytes
  – Which caches a global memory transaction passes through depends on GPU architecture and the type of access (read vs. write)
Global Memory Access Patterns

- When is a global memory read cached in L1?

<table>
<thead>
<tr>
<th>Compute Capability</th>
<th>Reads Cacheable in L1?</th>
<th>Cached by Default?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fermi (2.x)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Kepler (3.x)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Kepler K40 and later (3.5 and up)</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

- If the L1 cache is not used for global memory reads, the L2 cache is always used
Global Memory Access Patterns

- When is a global memory write cached in L1? Never

<table>
<thead>
<tr>
<th>Compute Capability</th>
<th>Reads Cacheable in L1?</th>
<th>Cached by Default?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fermi (2.x)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Kepler (3.x)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Kepler K40 and later (3.5 and up)</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

- Global memory writes are cached in the L2 cache
Global Memory Access Patterns

• Aligned and Coalesced Memory Access (*w/ L1 cache*)

• With 128-byte access, a single transaction is required and all of the loaded bytes are used
Global Memory Access Patterns

- Misaligned and Coalesced Memory Access (w/ L1 cache)

With 128-byte access, two memory transactions are required to load all requested bytes. Only half of the loaded bytes are used.
Global Memory Access Patterns

• Misaligned and Uncoalesced Memory Access (w/ L1 cache)

• With uncoalesced loads, many more bytes loaded than requested
Global Memory Access Patterns

• Misaligned and Uncoalesced Memory Access (w/ L1 cache)

• One factor to consider with uncoalesced loads: while the efficiency of this access is very low it may bring many cache lines into L1/L2 cache which are used by later memory accesses. The GPU is flexible enough to perform well, even for applications that present suboptimal access patterns.
Global Memory Access Patterns

• Memory accesses that are not cached in L1 cache are serviced by 32-byte transactions
  – This can improve memory bandwidth utilization
  – However, the L2 cache is device-wide, higher latency than L1, and still relatively small ➔ many applications may take a performance hit if L1 cache is not used for reads
Global Memory Access Patterns

- Aligned and Coalesced Memory Access (w/o L1 cache)

- With 32-byte transactions, four transactions are required and all of the loaded bytes are used
Global Memory Access Patterns

• Misaligned and Coalesced Memory Access (w/o L1 cache)

• With 32-byte transactions, extra memory transactions are still required to load all requested bytes but the number of wasted bytes is likely reduced, compared to 128-byte transactions.
Global Memory Access Patterns

• Misaligned and Uncoalesced Memory Access (w/o L1 cache)

• With uncoalesced loads, more bytes loaded than requested but better efficiency than with 128-byte transactions
Global Memory Access Patterns

- Global Memory Writes are always serviced by 32-byte transactions
Global Memory and Special-Purpose Memory

• Global memory is widely useful and as easy to use as CPU DRAM

• Limitations
  – Easy to find applications with memory access patterns that are intrinsically poor for global memory
  – Many threads accessing the same memory cell ➔ poor global memory efficiency
  – Many threads accessing sparse memory cells ➔ poor global memory efficiency

• Special-purpose memory spaces to address these deficiencies in global memory
  – Specialized for different types of data, different access patterns
  – Give more control over data movement and data placement than CPU architectures do
Shared Memory

- Declared with the `__shared__` keyword
- Low-latency, high bandwidth
- Shared by all threads in a thread block
- Explicitly allocated and managed by the programmer, manual L1 cache
- Stored on-SM, same physical memory as the GPU L1 cache
- On-SM memory is statically partitioned between L1 cache and shared memory
Shared Memory Allocation

• Shared memory can be allocated statically or dynamically

• Statically Allocated Shared Memory
  – Size is fixed at compile-time
  – Can declare many statically allocated shared memory variables
  – Can be declared globally or inside a device function
  – Can be multi-dimensional arrays

```c
__shared__ int s_arr[256][256];
```
Shared Memory Allocation

- Dynamically Allocated Shared Memory
  - Size in bytes is set at kernel launch with a third kernel launch configurable
  - Can only have one dynamically allocated shared memory array per kernel
  - Must be one-dimensional arrays

```c
__global__ void kernel(...) {
    extern __shared__ int s_arr[];
    ...
}
```

```c
kernel<<<nblocks, threads_per_block, shared_memory_bytes>>>(...);
```
Matvec using shared memory
Matrix Vector Multiplication

```c
/** N = 1024, 4 blocks, 256 threads/per block */
__global__ void
matvec_kernel_shared(float * A, float * B, float * C, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x; /* 0 - 1023 */
    int j;

    extern __shared__ float B_shared[]; /* the same size as B[1024] */
    B_shared[i] = B[i];
    /* for block 0: 0-255 are filled */
    /* for block 1: 256-511 are filled */
    /* for block 2: 512-767 are filled */
    /* for block 3: 768 - 1023 are filled */

    B_shared[(i+256)%1024] = B[(i+256)%1024];
    B_shared[(i+512)%1024] = B[(i+512)%1024];
    B_shared[(i+768)%1024] = B[(i+768)%1024];

    __syncthreads();

    if (i < N) {
        float temp = 0.0;
        for (j=0; j<N; j++)
            temp += A[i*N+j] * B_shared[j];

        C[i] = temp;
    }
}
```
Matrix Vector Multiplication

__global__ void matvec_kernel_shared_general(float * A, float * B, float * C, int N) {
    int i = blockDim.x * blockIdx.x + threadIdx.x; /* 0 - 1023 */
    int j;

    extern __shared__ float B_shared[];
    int k;
    for (k=0; k<gridDim.x; k++) {
        B_shared[(threadIdx.x + k*blockDim.x)%N] = B[(threadIdx.x + k*blockDim.x)%N];
    }

    __syncthreads();

    if (i < N) {
        float temp = 0.0;
        for (j=0; j<N; j++)
            temp += A[i*N+j] * B_shared[j];
        C[i] = temp;
    }
}
Constant Memory

- Declared with the `__constant__` keyword
- Read-only
- Limited in size: 64KB
- Stored in device memory (same physical location as Global Memory)
- Cached in a per-SM constant cache
- Optimized for all threads in a warp accessing the same memory cell
Constant Memory

• As its name suggests, constant memory is best used for storing constants
  – Values which are read-only
  – Values that are accessed identically by all threads

• For example: suppose all threads are evaluating the equation
  \[ y = mx + b \]
  for different values of \( x \), but identical values of \( m \) and \( b \)
  – All threads would reference \( m \) and \( b \) with the same memory operation
  – This broadcast access pattern is optimal for constant memory
• A simple 1D stencil
  – target cell is updated based on its 8 neighbors, weighted by some constants $c_0, c_1, c_2, c_3$

\[
f'(x) \approx c_0 (f(x + 4h) - f(x - 4h)) + c_1 (f(x + 3h) - f(x - 3h)) - c_2 (f(x + 2h) - f(x - 2h)) + c_3 (f(x + h) - f(x - h))\]
Constant Memory

- `constantStencil.cu` contains an example 1D stencil that uses constant memory

```c
__constant__ float coef[RADIUS + 1];

cudaMemcpyToSymbol(coef, h_coef, (RADIUS + 1) * sizeof(float));

__global__ void stencil_1d(float *in, float *out, int N)
{
    ...
    for (int i = 1; i <= RADIUS; i++) {
        tmp += coef[i] * (smem[sidx + i] - smem[sidx - i]);
    }
}
```
CUDA Synchronization

• When using shared memory, you often must coordinate accesses by multiple threads to the same data

• CUDA offers synchronization primitives that allow you to synchronize among threads
CUDA Synchronization

__syncthreads

- Synchronizes execution across all threads in a thread block
- No thread in a thread block can progress past a __syncthreads before all other threads have reached it
- __syncthreads ensures that all changes to shared and global memory by threads in this block are visible to all other threads in this block

__threadfence_block

- All writes to shared and global memory by the calling thread are visible to all other threads in its block after this fence
- Does not block thread execution
CUDA Synchronization

__threadfence__

- All writes to global memory by the calling thread are visible to all other threads in its grid after this fence
- Does not block thread execution

__threadfence_system__

- All writes to global memory, page-locked host memory, and memory of other CUDA devices by the calling thread are visible to all other threads on all CUDA devices and all host threads after this fence
- Does not block thread execution
Suggested Readings

1. Chapter 2, 4, 5 in *Professional CUDA C Programming*

